6-1-2020

Automatic Generation of Input Grammars Using Symbolic Execution

Linda Xiao
Dartmouth College

Dartmouth Computer Science Technical Report TR2020-898

AUTOMATIC GENERATION OF INPUT GRAMMARS USING SYMBOLIC EXECUTION

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Bachelor of Arts in Computer Science

Linda Xiao

Advised by Sean W. Smith

Department of Computer Science

Dartmouth College

Hanover, New Hampshire

June 2020

Abstract

Invalid input often leads to unexpected behavior in a program and is behind a plethora of known and unknown vulnerabilities. To prevent improper input from being processed, the input needs to be validated before the rest of the program executes. Formal language theory facilitates the definition and recognition of proper inputs.

We focus on the problem of defining valid input after the program has already been written. We construct a parser that infers the structure of inputs which avoid vulnerabilities while existing work focuses on inferring the structure of input the program anticipates. We present a tool that constructs an input language, given the program as input, using symbolic execution on symbolic arguments. This differs from existing work which tracks the execution of concrete inputs to infer a grammar. We test our tool on programs with known vulnerabilities, including programs in the GNU Coreutils library, and we demonstrate how the parser catches known invalid inputs.

We conclude that the synthesis of the complete parser cannot be entirely automated due to limitations of symbolic execution tools and issues of computability. A more comprehensive parser must additionally be informed by examples and counterexamples of the input language.

Acknowledgements

I would like to thank Professor Sean W. Smith for advising this thesis and always suggesting a giant literature of text to accompany every topic. I would also like to thank Professor Thomas H. Cormen for being a friend and mentor over the four years and basically operating under the assumption that I would write a thesis since freshman year. I am grateful for all the staff and faculty in the Department of Computer Science that filled the confusing halls of Sudikoff with familiar faces.

Additionally, I thank the other students in the Trust Lab for ever engaging lunch discussions and helping me navigate this field of research. Thank you to all my friends and family for your encouragement and support, especially towards the end when finishing up this project. Special thanks to Themis for being a sounding board for whenever I got stuck on a problem.

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0075. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

Contents

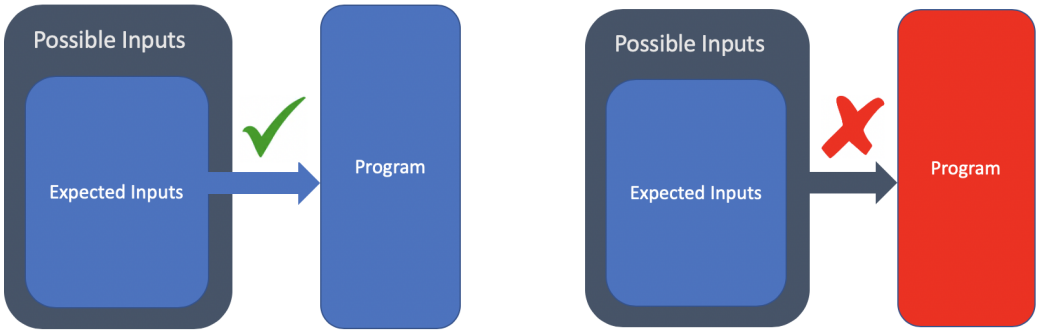
Chapter 1

Introduction

Section 1.1

Problem

Every computer program accepts a set of inputs and then returns a set of outputs. However, not every program has specified exactly what format of inputs it expects. Maliciously crafted input can cause the program to execute unexpectedly and lead to unintended consequences. It may cause the application to perform unauthorized actions like expose sensitive information or execute system commands. One such example is the Heartbleed attack, a bug in the OpenSSL implementation of the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols, designed to secure communications on a computer network [14]. It consists of a crafted heartbeat message, or a message to demonstrate responsiveness between two connected computers. In a heartbeat message exchange as intended, one computer sends a signal to the other, indicating the size of the message and expects the other computer to simply send the message back. Without verification of the contents of this message, specifically that the message actually consists of the number of bits that it claims, one computer can trick the computer into divulging additional



(a) The program receives expected input (b) The program receives unexpected input

Figure 1.1: The program will exhibit unpredictable behavior if it is given input it does not expect

information, like recently entered usernames and passwords.

The Heartbleed attack is only one example. The issue of input validation is rampant. A simple search for "input validation" on Common Vulnerabilities and Exposures (CVE), on May 16, 2020, reveals 1379 results, and "buffer overflow", a common form of improper input, reveals over ten thousand entries. Cyberattacks exploit vulnerabilities in programs that fail to parse and validate input, necessitating a stringent form of input validation.

As shown in Figure 1.1, the universe of possible input may differ from the input that program is designed to accept and operate on. Proper input validation relies on defining the accepted set of inputs and then developing a parser that accepts only this exact set.

1.1.1. Thesis Summary

This thesis aims to automatically generate a grammar that defends against detected vulnerabilities by only consuming the program as input. The grammar will be implemented as a parser modeling the same structure. The parser will then be prepended to the program to only allow valid input to execute through the program. Valid input is defined as strings within the formal language specified by the

grammar, and these inputs should not trigger known or detected vulnerabilities. The details of what a grammar entails first calls for a discussion of the underlying approach to handling input as a formal language in language-theoretic security.

Section 1.2

Background

1.2.1. Language-theoretic Security (LangSec)

Programmers are advised not to trust input data, but these terms are ambiguous. Parsers are the first line of defense with validating input to ensure that preconditions are met before executing the code. Parsers break the input into conveniently packaged elements, which the rest of the code processes. However, they often fall victim to improperly checking preconditions assumed by the rest of the code and receiving unexpected input, leading to unpredictable behavior like buffer overflows or memory corruption. Often times a series of *if* statements check the input and fail to completely validate a program's assumptions about the input. In addition, it is difficult to verify whether the input handling correctly accepts or rejects valid and invalid input. For complex enough input formats, this is computationally undecidable. Without strict guidelines of what a valid input entails, developers are more likely to skimp on checking whether or not the data conforms to expectation.

Language-theoretic security is based on applying formal language theory to define input to secure communication with a program. A formal language is defined as the set of strings over a finite alphabet. A grammar characterizes a language with a set of rules that decides which strings, formed by elements in the alphabet, are part of the language and which are not. The size of a language can be, and often is, infinite.

The premise of language-theoretic security states that input handling should

treat the set of all valid inputs as a formal language and construct a parser that recognizes this language [25]. A proper program must recognize input correctly, so each program implicitly has a set of expected inputs. LangSec declares that this set of inputs needs to be made explicit in a computational-theoretic and formal language-theoretic approach by detailing a grammar to describe it. In addition, the grammar needs to be as simple as possible with simplicity described by its position in the Chomsky hierarchy [17, 26]. The complexity of languages ranges from regular expressions to Turing-recognizable languages, each requiring increasingly complicated machines to recognize strings in the language.

Momot et. al [23] details the standard process to input processing and protocol design. First, define the set of acceptable inputs to a program via a grammar or formal language, ensuring the grammar is as simple as possible on the Chomsky scale of syntactic complexity. Second, construct a parser plainly following this grammar. Finally, establish a clear boundary between input validation and processing by only processing the input after it is validated by the parser. Invalid input should be rejected immediately and never processed in the first place. This will protect against shotgun parser bugs like Heartbleed and Android Master Keys bug, where input validation is mixed with processing code.

The implementation of the grammar must derive and resemble the grammar itself. Parser combinator toolkits, like Hammer [23], facilitate the design of this parser. Parser combinators combine individual parsers to output a new parser. Parser combinators consist of primitives and combinators. Primitives specify the basic unit of a token to be parsed like end of line tokens, strings, integers and individual character tokens in a range, as shown in Table 1.1. Combinators combine these tokens to compose parsers like in sequence or repetition, as shown in Table 1.2. Hammer's combinators resemble grammar combinators like concatenation and

Syntax	Usage	Description
`h_ch`	`h_ch('a')`	Matches a single specified character token
`h_ch_range`	`h_ch_range('a', 'z')`	Matches a single token in the specified character range
`h_token`	`h_token("hello", 5)`	Matches the specified string of given length
`h_int64`	`h_int64()`	Matches a signed 8-byte integer
`h_end_p`	`h_end_p()`	Matches the end of the parser input

Table 1.1: Hammer parser primitives

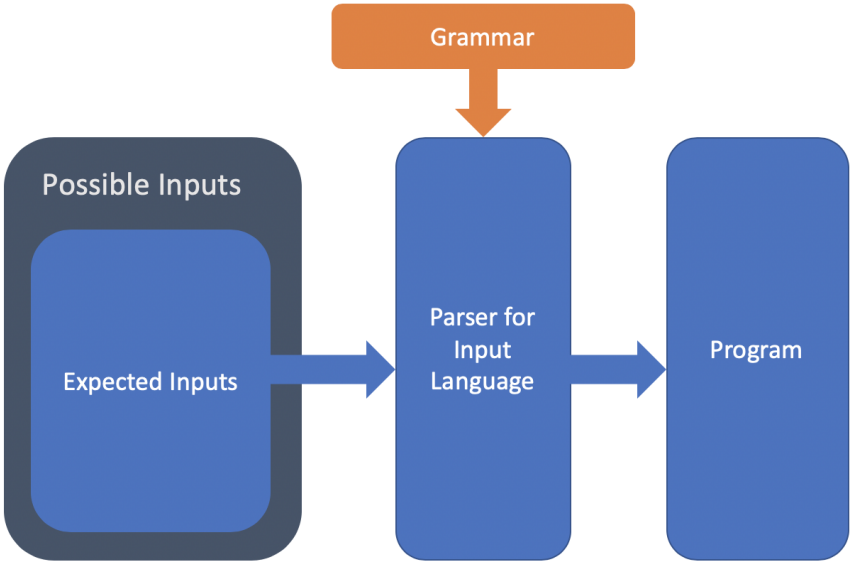
Kleene star, so the grammar is easily recognizable from the parser implementation. The syntax of Hammer parsers is also easy to interpret.

Figure 1.2 depicts the series of events involved in validating input with LangSec. The construction of a LangSec parser first requires defining the grammar. Ideally the definition of this grammar occurs in the writing of the program, but existing legacy code would also benefit from proper input handling. The grammar is then implemented via a parser combinator library like Hammer. Every input must first go through the parser and only valid, accepted input continues to be executed by the program. Invalid input should be immediately rejected by the parser and never be processed by the program.

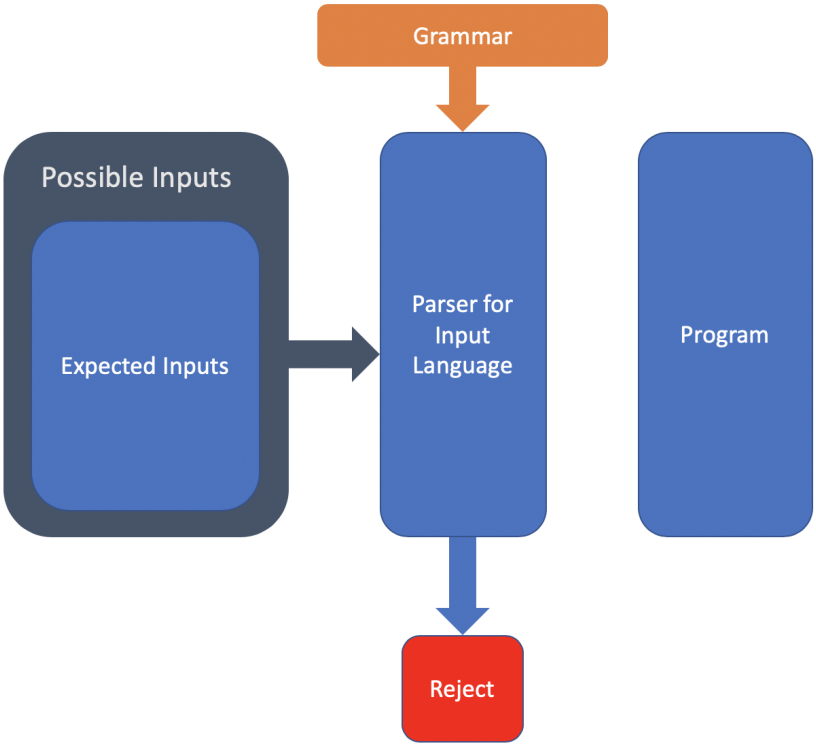
The parsing of input currently requires manual intervention to construct the grammar. Unless the developer has a formal and written out a grammar a priori, extracting the formal language of a data format from a written program or data examples is a very hands-on process. Programs often do not have a detailed specification of expected input and rely on parsing scattered throughout the program.

Syntax	Usage	Description
h_choice	h_choice(h_ch('a'), h_ch('b'), NULL)	Performs the Boolean "or" operation
h_sequence	h_sequence(h_ch('a'), h_ch('b'), NULL)	Performs the concatenation operation
h_repeat_n	h_repeat_n(h_ch('a', 5))	Repeats given parser n times
h_many	h_many(h_ch('a'))	Performs the Kleene star operation for 0 or more repetitions
h_many1	h_many1(h_ch('a'))	Performs the + operation for 1 or more repetitions
h_optional	h_optional(h_ch('a'))	Performs the ? operation to match 0 or 1 occurrences

Table 1.2: Hammer parser combinators



(a) The parser accepts expected input



(b) The parser rejects unexpected input

Figure 1.2: The parser implements the grammar. The language the parser accepts should be equivalent to the language the program expects. If the parser rejects an input, the program never processes it.

The goal of this thesis is to computationally automate this process by taking a program as input and outputting a grammar.

Language-theoretic security establishes validating input as central to any design process. By restricting the complexity of the input language, it is possible to design parsers with formally verifiable properties. However, many existing programs, especially those designed without security in mind, lack a robust input handling procedure. LangSec starts with defining a grammar of the input language, but for such legacy programs, the grammar is neither explicit nor clearly extractable. I set out to automatically derive a formal language parser for the input by just looking at the program and employing the tool of symbolic execution to follow its execution pattern.

1.2.2. Symbolic Execution

The ultimate goal of a testing technique is to detect or rule out vulnerabilities in a program. One approach would be to test the program using distinct, random inputs, but there is no guarantee these inputs will trigger a vulnerability if one exists. Symbolic execution provides a more comprehensive approach that explores many execution paths at the same time and checks if each path is exploitable.

Symbolic execution is a software testing technique that executes a program on symbolic input, which is input that's allowed to be anything [9]. Static analysis tools attempt to find problems before they happen without running the code, but they often require an expert in static analysis to distinguish false positives and real bugs. On top of that, executing a piece of code helps identify certain classes of errors such as functional correctness. Symbolic execution is easy to use as a bug finding tool. It evaluates the program on symbolic input values, as opposed to concrete values. When the program reaches a branch in execution, it first checks the feasibility of each of the branches, then duplicates the state and traverses each

of the branches separately. It accumulates constraints for the input that follows the specified path. Once the path terminates or finds a bug, it uses an automated theorem prover to provide a concrete value that satisfies these constraints. The concept can be clarified with an example, shown in Figures 1.3 and 1.4.

Figure 1.3: Symbolic Execution Sample Program

```
1    int foofum(int x, int y) {
2      int z = x + y;
3      assert(z != 0);
4      if (z <= 10) {
5        return z / y;
6      }
7      return z;
8    }
```

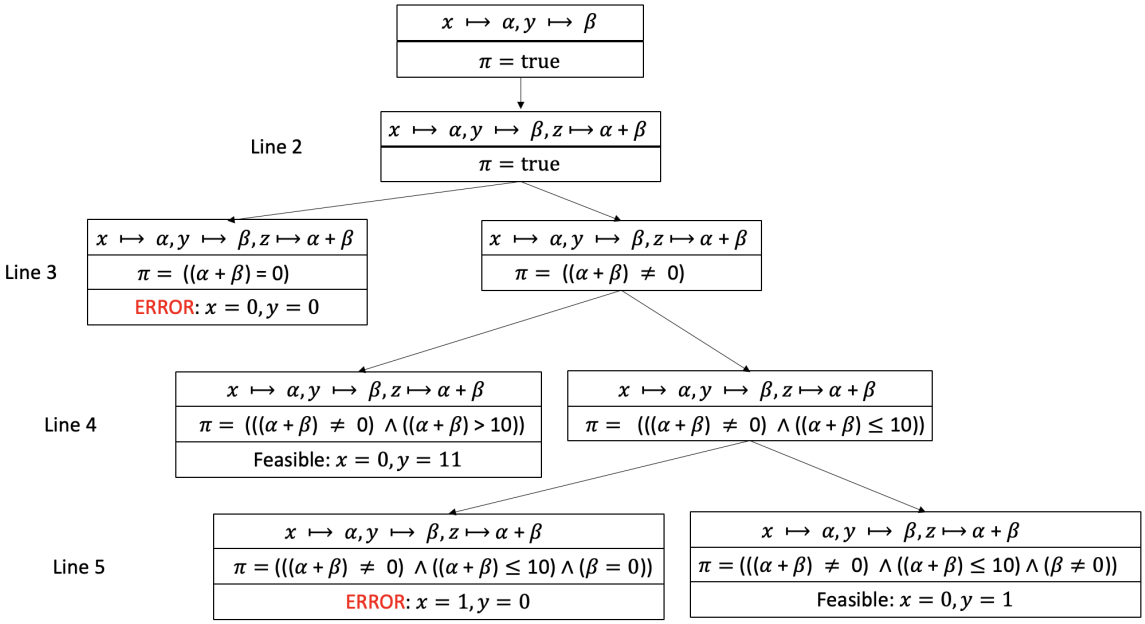


Figure 1.4: Execution tree for example in Figure 1.3 formed by branching the path at each dangerous operation. The constraints generated and the order of traversal may differ based on the symbolic execution tool used.

We walk through the code and tree step by step. The program is executed with two symbolic values, x and y, meaning their values can be anything. π maintains

the constraints that lead up the state. In the beginning, there are no constraints, so π evaluates to true. The code branches at any potentially dangerous operation. Line 2 forms z by adding the two symbolic values. For the assert statement on Line 3, the symbolic executor will solve the current constraints to see if there exists a value that evaluates the assert expression to false. If there is, it terminates with an error and details both the path condition and an example of an input that triggers it. It also branches for when the assert statement returns true. Similarly, the path branches for the *if* statement in Line 4. If this statement evaluates to false, the program immediately ends. Otherwise, the division of z and y in Line 5 is a dangerous operation because at this point, the path condition has not forbidden y from being zero. The path branches and every path terminates. When a path terminates, a solver solves the constraints to produce a concrete example that followed the path.

We use the open source symbolic execution library KLEE, an industrial-grade symbolic execution library for C code that runs on LLVM bitcode [5]. KLEE has been used to detect bugs in version 6.10 of the GNU COREUTILS library as well as BUSYBOX and MINIX [9].

Although symbolic execution intelligently executes code to perform an exhaustive search of possible input, it falls victim to a few limitations. Introduced in the 70s [19], symbolic execution has become more feasible due to better theorem provers (SAT/SMT solvers), more computational power, and heuristics to control the exponential explosion of paths. Symbolic execution shows promising coverage in research, but in practice, it is difficult to consistently achieve high coverage. Common concerns include the exponential explosion of the number of paths, the limitations of constraint solvers, and the difficulty with modeling the environment in which the code is run like system or library calls. However, symbolic execution is still a useful testing tool for many programs.

As shown by Cadar et al [9], klee achieves high code coverage on a diverse set of real, complicated programs. The ability to create symbolic files models the file system of the environment. While these results may be difficult to generalize to all programs, the experiments of klee demonstrate the potential of symbolic execution.

klee's goals are to (a) Reach every line of executable code and (b) Detect at each dangerous operation (e.g., deference, assertion) if any input value exists that could cause an error [9]. It makes this feasible through several optimizations like compact state representation, query optimization and state scheduling. Assuming these two goals are met, we aim to achieve complete coverage of inputs that may trigger vulnerabilities.

The klee constraint language is KQuery, described in the KQuery documentation. The constraints capture the meaning of the π or path condition, as exemplified in Figure 1.4. Running klee on the simple example gave four paths, with constraints exactly as described. For the constraints in Line 5, we observe that we can perform several optimizations, which klee enacts. For example, it simplifies the path of the left box in Line 5 with the knowledge that $\beta = 0$:

$$((\alpha + \beta) \neq 0) \wedge ((\alpha + \beta) \leq 10) \wedge (\beta = 0)$$

becomes

$$(\alpha \neq 0) \wedge (\beta = 0) \wedge (\alpha \leq 10)$$

This is an example of the query optimization that klee performs.

We will use symbolic execution to generate constraints that lead to vulnerabilities. We will then construct a parser out of these constraints. In the simple example, this parser would reject all input where $\beta = 0$ and when $\alpha + \beta = 0$. Naturally, the

parser is predicated on the symbolic executor's ability to detect dangerous operations. If it is unable to detect a vulnerability that we are aware of, we can add an assert statement that evaluates to false for the given vulnerability so that klee generate those constraints.

Section 1.3

Proposed Solution

LangSec asserts that security is only guaranteed when input is validated as a formal language. LangSec highlights the importance of designing a simple input language for protocols and programs in general, which leads to a parser implementation of this grammar. However, while these principles facilitate the design of new applications and protocols, we want to apply LangSec's principles to existing programs.

We set out to automatically construct this grammar from the program itself by using symbolic execution. The grammar defines inputs that do not trigger certain vulnerabilities. We are limited to the bugs that symbolic execution can find or that has already been reported, so we focus on constructing a grammar that rejects input that triggers those specific vulnerabilities. The goal is to take a program as input, and output a parser that protects against detected or known vulnerabilities. This parser then gets prepended to the program and rejects any invalid input, which is then never executed in the program.

Section 1.4

Related Work

I examine other works related to extracting an input format from the program itself and point out their shortcomings.

1.4.1. Reverse Engineering Protocols

Most of the literature around inferring an input format centers around reversing message formats for protocols. To the best of our knowledge, no previous work integrates formal language theory and symbolic execution. A section of literature focuses on the automatic extraction of the protocol message format without access to the protocol specification [7, 8, 11, 12]. The message format consists of sequences of fields organized with certain rules of what values each field can take within a fixed domain. Caballero et. al [8] looks at program binaries. Cui et. al and Caballero et. al observe execution and network traces to extract the message format and field semantics [7, 11, 12]. These papers focus on a subset of the general problem for extracting the input format for a protocol rather than an arbitrary program. Protocols introduce a different challenge because depending on its position in the protocol state machine, different message formats will be expected. None of the aforementioned papers address how they would extract this protocol state machine. To be able to extract the protocol message format, we need two languages: one for the protocol messages and another for the protocol state machine. Antunes et. al attempts to reconstruct both languages from the network traces [4]. We focus on a different problem, not specific to protocol design, so there implicitly is a single state for the program.

1.4.2. Learning Languages from Examples

The more theoretical side of related work focuses on deriving a formal grammar. In [15], Gold established the limitations of learning a language from examples in the language. By simply presenting positive examples, Gold concluded that infinite regular languages are not learnable. This result can be extended to more complex languages. Angluin demonstrated that an infinite regular language can be constructed from examples and counterexamples [3]. A counterexample is a string that belongs in the conjectured set but not in the correct set. This result depends on having a teacher that can answer membership queries of whether or not a string belongs in the language. The teacher must also say yes if the conjectured language is equal to the desired language and provide a counterexample otherwise. The method of learning based on examples and counterexamples is difficult to implement in practice because such a teacher/oracle does not exist that can assert when the conjectured language is equal to the desired language.

1.4.3. Synthesize Input Grammar from Execution

Finally, we look at more recent work to engineer an input grammar based on the principles of LangSec. In [13], Curley uses machine learning on a dataset of URIs from an Apache HTTP access log to extract a context free grammar. Curley demonstrates where theory differs from practice by using a recurrent neural network to both predict the response codes and generate strings in the language. Curley concludes that while it may be impossible to prove that all context free languages can be learned from examples and counterexamples, certain languages in this set can be learned with high accuracy. However, most programs lack an elaborate database of tagged strings, and it is unreasonable to train a deep learning model for every input language. Although the model implicitly recognizes the grammar, there is

no way to extract the grammar out to prove its accuracy.

Bastani et. al designed GLADE, which [6] builds a context free grammar from a set of input examples and blackbox access to the program. Starting with inputs, GLADE constructs an increasingly general language based on repetition, alternation and recursive constructs to the language. However, it does not take advantage of the program's structure and does not detail how to recover the readable grammar.

Autogram [18] mines context free input grammars by observing how an input is processed in a program and tags each piece of stored data with the input fragment it comes from. It then outputs the grammar in Backus normal form. It achieves 100% coverage of the language in many cases like with Apache Commons CSV and JSON Objects by analyzing the Minimal JSON library. To perform dynamic analysis, Autogram begins with a set of inputs that belong to the grammar, which may or may not be feasible depending on whether a training set or reference grammar exists. On top of that, it runs on parsers for commonly used file formats rather than an arbitrary piece of code. Its goal is to capture and formally define the parser's grammar while our goal is to output a grammar that protects against certain vulnerabilities.

Autogram and mimid [16,18] form the literature of constructing a human readable context free grammar from input. mimid improves upon Autogram's dynamic analysis by inserting trackers in the source program at conditions, loops, and method entries and exits. These trackers follow input character accesses being made to determine which method calls should be associated with the particular character index, regardless of whether these characters are stored. In contrast, Autogram operates on the data flow of stored variables. mimid constructs a parser tree, which generalizes to grammars, from methods accessing some part of the in-

put. Mera also uses taint analysis to further refine automatically generated context free grammars to express semantic relationships between grammar elements [22]. For example a context free grammar can specify that the field maxsum is an integer, but the value of maxsum may restrict the values of other fields of integers.

Among these different tools to infer input grammars, this thesis addresses the dissimilar but related problem of inferring a grammar that protects against detected vulnerabilities. Our work also analyzes inputs to all programs, not just programs that are parsers. While further analysis of their outputted grammar may lead to insight about possible exploits, we focus directly on the problem of preventing these exploits. Instead of returning just the grammar, we return a parser which implements the grammar.

Section 1.5

Overview

I set out to automatically infer an input grammar from a program. The grammar and its parser implementation only accept inputs that do not trigger certain vulnerabilities.

I will first detail the method of deriving the grammar and parser from the symbolic execution constraints in Chapter 2. Then, I apply these methods on a collection of code samples, including the heavily tested GNU Coreutils library in which klee detected undiscovered bugs in Chapter 3. Our tool generates parsers for each symbolic input. Chapter 4 discuses the limitations of our tool and future work to address these limitations as well as other directions to take to advance the tool. Finally, in Chapter 5, we summarize contributions made and the practicality of our tool.

Chapter 2

Methods

In this section, we first provide an overview of the architecture, then describe the main steps in more detail.

The goal of this paper is to automatically generate a parser combinator implementation of the input grammar that protects against certain vulnerabilities. To do this, we use symbolic execution.

Figure 2.1 provides a visual overview of the steps used to generate the parser. The sections highlighted in orange denote my own contributions. KLEE takes the LLVM bitcode of the program and symbolically executes the file with the symbolic arguments provided. The user must specify the size, number, and type of symbolic inputs. If the user knows of an input that triggers the vulnerability, they can model the symbolic arguments based on this input. The `klee-stats` tool provides the line coverage of an execution of KLEE on provided symbolic inputs. If the line coverage is not satisfactory, the user can allocate more symbolic arguments, increase their sizes or allocate different types of arguments. Then, they run KLEE with these new parameters, and repeat the process of checking line coverage and rerunning if necessary. Finding the proper allocation of symbolic arguments is an iterative process that can be sped up with knowledge of what inputs can trigger different

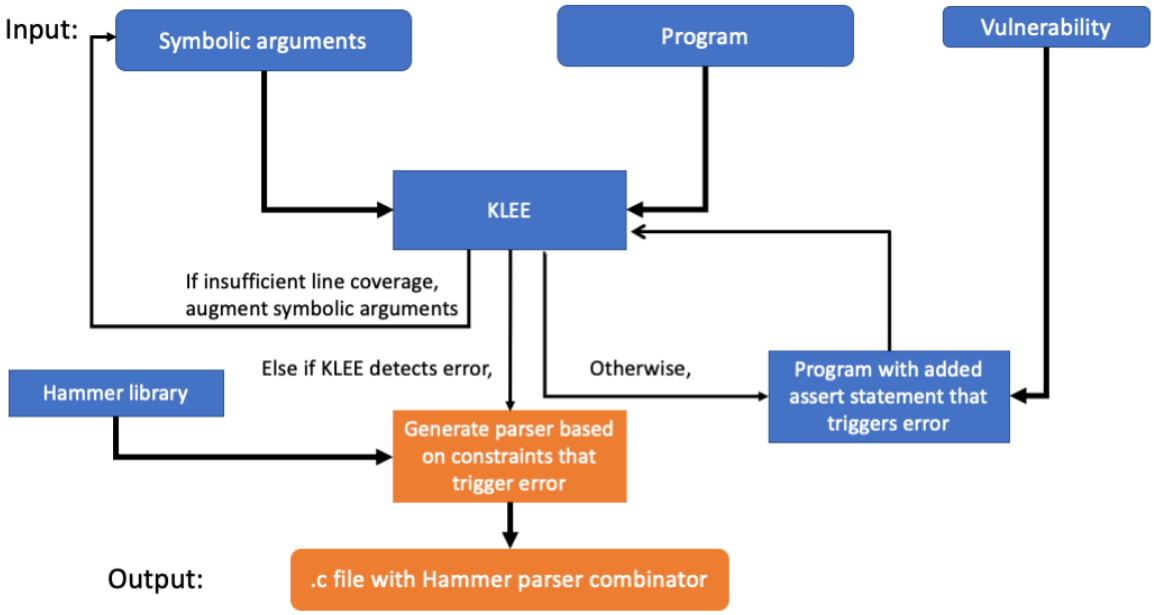


Figure 2.1: The envisioned overview of how we automatically generate the grammar. The bold arrows represent necessary inputs or libraries. The standard arrows represent conditional splits.

behaviors in the program.

KLEE detects errors such as pointer, free, abort, assert or division errors at dangerous operations. For every error that KLEE detects, it generates the constraints that led to the error. We use KLEE's constraint parser to parse the constraint written in the .kquery file into an Abstract Syntax Tree. The KQuery documentation elaborates the KQuery format, as noted in Section 1.2.2. We then traverse this tree to generate a parser that invokes the Hammer library parser combinators. If KLEE does not detect a known vulnerability, we place an assert statement that evaluates to false in the path that triggered the error, so that KLEE identifies it as an assert error. We then run KLEE on the updated program with the additional assert statement. An assert false statement can be inserted in a section of code that should not run, and KLEE will determine the constraints that lead to that section. We traverse the constraints to generate a parser. The final output is a .c file with the implementation of the parser for each symbolic argument.

Figure 2.2: Sample buffer overflow vulnerability

```
1   int main(void) {
2       char buff[4];
3
4       printf("\n Enter the password : \n");
5       gets(buff);
6
7       ...
8       return 0;
9   }
```

Throughout this section, we demonstrate the architecture with a simple example of a buffer overflow in the `gets` C function, shown in Figure 2.2. The C file consists of this main function and is called password.c. The `...` represents code that compares the inputted string with the actual password and grants access if it is correct. The `gets` function is known to be unsafe because it does not verify if the input can fit inside the buffer. A buffer overflow will affect the later code that validates the input and lead to unexpected behavior like potentially granting disallowed access.

Section 2.1

Running KLEE

To run KLEE on a program, we must first extract its LLVM bitcode by either compiling it to emit LLVM or extract-bc from the executable.

Then, we run KLEE with symbolic inputs. KLEE allows specifying symbolic arguments, files, stdin and stdout as well as their size. We illustrate examples with different types of symbolic arguments later. For the `gets` example, we first extract the bitcode from the file with

```
clang -emit-llvm -g -c password.c -o password.bc
```

We then run klee on this bitcode file with a symbolic stdin, meaning the stdin can take any value. We make the stdin symbolic because our code takes input from standard input, and we limit it to an arbitrary 100 characters.

```
klee --libc=uclibc --posix-runtime password.bc -sym-stdin 100
```

`uclibc` provides definitions for all the external functions the program may call, and the command tells klee to load that library and link it with the application before it starts execution. `posix-runtime` works with klee and the `uclibc` library to provide the majority of the operating system facilities used by command line applications like `write`. Without these libraries, klee is unaware of what the `gets` function does.

For other programs, we similarly specify the symbolic arguments for which we want to generate parsers when running klee.

If klee detects an error while running, it creates a test file for that error and continues running on other constraints. For programs with known vulnerabilities that klee does not detect as an error, add an assert statement that evaluates to false at the known error and run klee again. Running klee on a known invalid input without any symbolic arguments generates an error file that reveals the line the error is thrown. To make sure this error is not a false positive, we may replay the input that triggered the error with `klee-replay`. False positives are theoretically impossible because the input follows the same path as klee in the unmodified program. However, non-determinism, bugs and heuristics used in klee may lead to such errors [9]. The syntax of `klee-replay` operates on the generated .ktest file. The ktest file contains the result of applying a constraint solver to output an input that satisfies these constraints. Here is an example of running `klee-replay`:

```
klee-replay ./password ./klee-out-1/test000011.ktest
```

Figure 2.3: Constraint for pointer error in gets

```
1   array stdin[100] : w32 -> w8 = symbolic
2   (query [(Eq false
3               (Eq 10
4                   (Extract w8 0 (ZExt w32 (Read w8 0 stdin)))))
5           (Eq false
6               (Eq 10
7                   (Extract w8 0 (ZExt w32 (Read w8 1 stdin)))))
8           (Eq false
9               (Eq 10
10                  (Extract w8 0 (ZExt w32 (Read w8 2 stdin)))))
11          (Eq false
12              (Eq 10
13                  (Extract w8 0 (ZExt w32 (Read w8 3 stdin)))))]
14         false [] [stdin])
```

For our example, KLEE generates 11 paths and tests, the last of which led to a pointer error. In the next section, we look at the constraints for this path.

Section 2.2

Parsing Constraints

After running KLEE, we look at the constraints that trigger certain errors. KLEE generally solves these constraints to identify an example for every possible path, but we want to capture the complete language of the constraints, not just one example. These constraints are textually represented in KQuery format, designed to be compact and easy to read and write. We walk through the constraint that generated the pointer error in our example as shown in Figure 2.3. We removed parts of the constraint that are not relevant to the stdin symbolic argument.

Each line in the constraint is either an array declaration (Line 1) or a query command (other lines). A query command is run by the constraint solver and is denoted by the keyword "query". The important parts of this query are the

constraint list and then the last element. We manually added `[] [stdin]` after the last `false` to specify that we want to generate a parser for this specific symbolic input (stdin).

The first line in Figure 2.3 declares stdin to be a symbolic buffer of size 100 storing 1 byte elements. The second line denotes a query and the start of a constraint list. Each line of the constraint is very similar, so we look at one line.

```
(Eq false (Eq 10 (Extract w8 0 (ZExt w32 (Read w8 0 stdin)))))
```

We start with the most interior layer of `Read w8 0 stdin`. This reads the 0th index of the symbolic array of size 8 bits. `ZExt` evaluates the lowest 32 bits of this value. `Extract` extracts 8 bits from this value starting from bit 0. After `ZExt` and `Extract`, the value is the same as the original 8 bits read. Then, the constraint declares that this values should not be equal to 10. This is repeated for each indices 1, 2 and 3.

The vulnerability was that we only allocated a buffer of size 4, but allowed arrays larger than size 4 to be written to it which may lead to a buffer overflow. `gets` stops reading at a newline character or when the end-of-file (EOF) is reached. If none of the first 4 characters are newline characters or EOF, we get a buffer overflow. This is indicated by our constraints. Index 0 to 3 must exist, so they are not EOF, and they are not the newline character, leading to an error.

Since KLEE's constraint solver operates on this KQuery language, the KLEE toolset already has a parser that consumes the constraint language and outputs an Abstract Syntax Tree (AST). Our tool then parses this tree to construct a Hammer parser that accepts the constraint language. Although we write the parser to accept inputs that give us an error, we can flip the logic to reject input that it parses successfully and accept input that it fails to parse. This flip is only provably computationally feasible if the language is within deterministic context free languages, so our parser

is limited to these languages.

We observe that klee operates on its symbolic input as an array of characters, and we take advantage of this format in designing the parser. We introduce a data structure that we call SetCombinator to store an intermediate representation of the constraints that then gets translated to Hammer invocations.

We set up the parser character by character and restrict the values each character can be. A SetCombinator is organized as an array of sets where each set represents the set of allowed characters from 0 to 127, the range of ASCII values. The unit size of an element of klee's arrays is always 8 bits. Representations of more bits are formed by concatenating multiple elements in the array. The size of the SetCombinator depends on the size of the symbolic input. Every time an index is `Read` from the symbolic array and certain constraints are applied, the respective index in the SetCombinator also updates its set to accommodate these constraints. The final set left at each index represents the intersection of every constraint applied to the index.

Going back to our `gets` example, we first initialize each set in our SetCombinator of size 100 with a complete set of $\{0, 127\}$. The process of constructing the SetCombinator is depicted in Figure 2.4. As we parse each constraint in the AST, we exclude 10 from the sets at index 0 to 3. We use $U = \{0, 1, ..., 127\}$ to denote the universe of values for each element in the array. The resulting SetCombinator looks like Table 2.1. Each index of our SetCombinator corresponds with the index in the symbolic array. We have restricted the first 4 indices to not included the newline character, encoded with ASCII value 10, and there are no restrictions on the other indices.

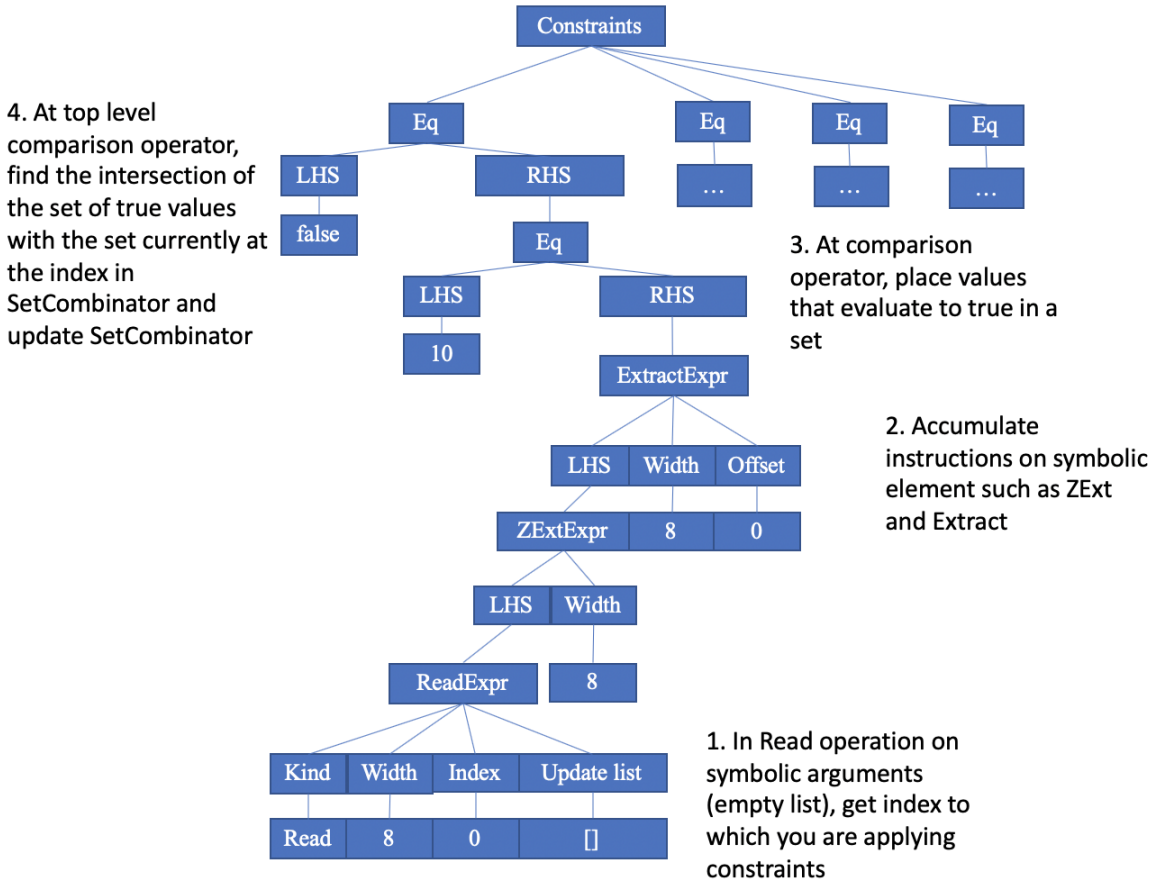


Figure 2.4: Steps to construct the SetCombinator for the `gets` example constraints. The SetCombinator representation is then turned into a parser implementation.

0	1	2	3	4	5	...	100
{0,9} ∪{11, 127}	{0,9} ∪{11, 127}	{0,9} ∪{11, 127}	{0,9} ∪{11, 127}	U	U	...	U

Table 2.1: SetCombinator Example

Section 2.3

Outputting Hammer Invocations

Once we have the SetCombinator filled out from the constraints, we can translate it to Hammer invocations. The Hammer parser will ultimately be a sequence of

elements where each element will represent the range of values the character is allowed to take. The parser will take strings as input, or sequences of characters.

Each index in the SetCombinator represents a range of possible ASCII character values. This can be captured with the `h_option` and `h_ch_range` or `h_ch` Hammer invocations, as described in Figures 1.1 and 1.2. These characters are then concatenated in an `h_sequence` or left as is if there is only one element in the sequence.

Running our algorithm on the KQuery file from the example, we get the parser

```
h_repeat_n(h_choice(h_ch_range(0,9), h_ch_range(11,127), NULL), 4)
```

Note that our algorithm will simplify repetitive indices by employing `h_repeat_n` and cut off the rest of the SetCombinator if there are no further restrictions on the later indices. If there are restrictions for every index, we end our parser with `h_end_p` to indicate that the parsed string should end there and makes sure that there is no input left to parse. Our parser stops at index 4 because the first 4 indices are the only characters whose values lead to the buffer overflow. The first 4 indices either belong in the character range 0 to 9 or 11 to 127, only excluding 10 from its range.

The actual output of this parser is a .c file which implements this parser in the main function. It parses stdin, but this can be modified to parse passed arguments as well. We chose to output the parser textually rather than by reference to allow the user to identify what makes up the parser and customize it as necessary.

The final step comes with compiling this Hammer implementation and attaching the parser to the start of the program as shown in Figure 2.5. The logic is a little reversed because the parser accepts inputs that would throw an error, so we negate the results. Inputs that trigger the error are blocked by the parser and never passed to the rest of the program.

We tested our parser on a few examples that would overflow the `gets` buffer and

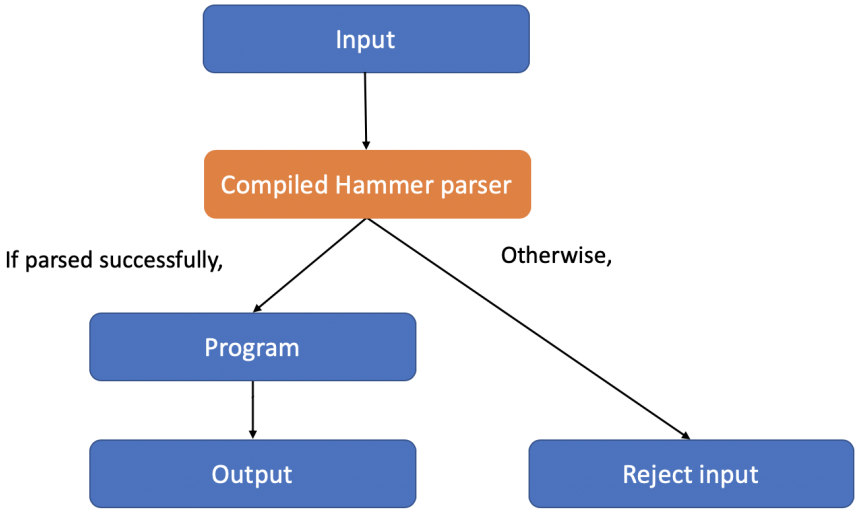


Figure 2.5: Prepending the parser to the program, so that the program never runs on invalid input

those that would not. It successfully rejects and accepts these inputs. This is not a complete testing method, but upon observation and knowledge of the behavior of `gets`, we can observe that the parser recognizes strings that overflow the finite buffer.

Section 2.4

Code and Documentation

The code was implemented as an additional option in the KLEE library, similarly to its tools like `kleaver`, which is their constraint solver and `klee-replay`, which replays generated test inputs. It currently consists of 921 lines of additional code. It is available on github.

Chapter 3

Experiments

Upon running the experiments, we adjust our architecture diagram to reflect limitations in purely generating parsers from the constraints, as indicated in Figure 3.1. The main difference lies after generating the parser combinator. Due to limitations in symbolic execution, the grammar often does not provide a notion of all the inputs that trigger the known errors. The parser needs to be manually configured after testing a series of examples and counterexamples based on the current grammar and observing the program's execution. After updating the parser and grammar, we repeatedly test examples and counterexamples until reaching a grammar that logically triggers all known errors. Knowledge of the program behavior would advise this manual testing process.

The ultimate goal is to reject inputs the trigger certain vulnerabilities. We first generate our parser to accept the language of invalid inputs, as described by the constraints. To reject invalid inputs, we negate the results of our parser. The complement of our languages is computable because our parser implements deterministic context free languages.

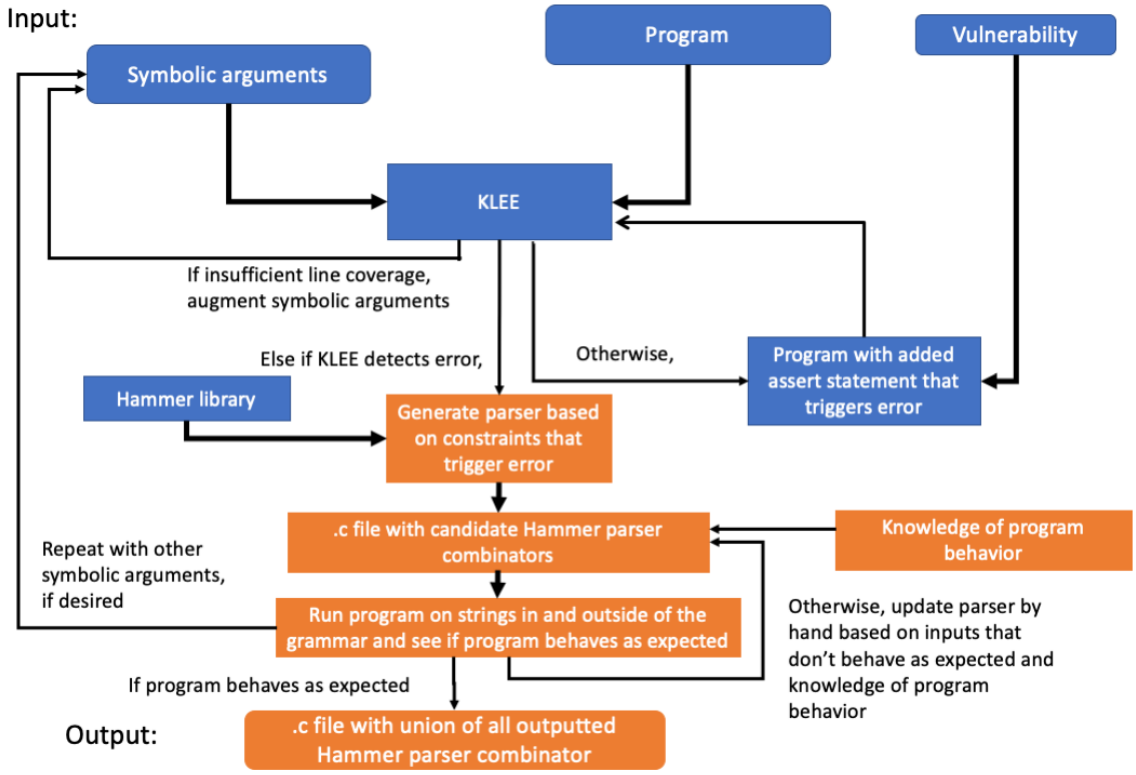


Figure 3.1: The overview of how we automatically generate the grammar, in practice. Orange represents our contributions. The bold arrows represent necessary inputs or libraries. The standard arrows represent conditional splits.

Section 3.1

GNU Coreutils 6.10

In its inaugural paper, klee reported detecting 10 errors in the GNU Coreutils 6.10 library. The GNU Coreutils library implements basic tools used in Unix-like operating systems such as `ls` and `cat` [2]. We run the same version on a 64-bit Ubuntu 18.04.4 Desktop. The paper ran the programs on a 32-bit machine, so the constraints generated on our machine are often more complex and memory intensive. We generate constraints for 5 of the 10 errors. Out of the other 5, we were only able to reproduce the error on the given input for 3 of the programs, but were unable to generate constraints for these errors. The length of the symbolic input for two of these three programs were over 26 characters, and running this exceeded the max run time allocated. For the past program, we were unable to reproduce an error on klee.

We reproduce in Table 3.1 the inputs that cause program crashes from [9], and we generate a parser that catches these inputs. Again, we only look at the set of 5 programs for which we were able to 1) reproduce the error by crashing the program on a known malformed input and 2) run klee to detect an error and thus output the constraints that triggered the error.

We ran klee with varying numbers and sizes of symbolic arguments. Because we had the additional insight of an example of an input that triggers the vulnerability, we modeled the symbolic input to follow that format.

We walk through the example of `mkdir`. Although we used several additional options like running it in a sandbox directory and maxing out the time the run took, we highlight the key options used with running klee. First, we run klee with

`mkdir -Z a b`
`mkfifo -Z a b`
`mknod -Z a b p`
`md5sum -c t1.txt`
`seq -f %0 1`
t1.txt: "\t \tMD5("

Table 3.1: A subset of the inputs that cause vulnerabilities in GNU Coreutils 6.10 [9]. This subset reflects the programs for which we were able to generate constraints that triggered errors. When replaying the values from solving the constraints, the values crashed the program.

```
klee --optimize --libc=uclibc --posix-runtime ./mkdir.bc
--sym-arg 2 --sym-arg 1 --sym-arg 1
```

The `--libc=uclibc` and `--posix-runtime` tags are the same as described in Section 2.1. The option `--optimize` optimizes `mkdir` program's code before execution. The `--sym-arg` options create three symbolic arguments, with the first one being 2 characters long, followed by one character and another character. After running klee, the .kquery file only detailed constraints for the 0th arguments (`arg00`) and the 2nd argument (`arg02`), implicitly implying the existence of a 1st argument with no constraints. We add `[]` and `[arg00 arg01 arg02]` to the last line before the parentheses, just as in Figure 2.2, to denote that we want parsers for our symbolic inputs `arg00, arg01` and `arg02`. The `[]` tells us there are no additional constraints on these arguments. Our tool outputs three different parsers for each of the symbolic arguments as shown in Table 3.2.

We rewrite the results in Table 3.2 by substituting the ASCII decimal values

arg00	h_sequence(h_ch(45), h_ch(90), NULL)
arg01	h_ch_range(0, 127)
arg02	h_choice(h_ch_range(0, 44), h_ch_range(46, 127), NULL)

Table 3.2: Hammer invocations automatically output by our tool to parse each of the symbolic arguments. Arguments are numbered from 0 to 2. Table 3.3 captures this table except it substitutes the ASCII values with ASCII characters, and introduces additional notation.

arg00	h_sequence(h_ch('-'), h_ch('Z'), NULL)
arg01	h_ch_range(0, 127)
arg02	h_not_ch('-')

Table 3.3: Substituting Table 3.2 parser ASCII character values with actual characters. The first symbolic argument must be -Z, followed by a nonempty argument, followed by an argument that is not -.

with their respective characters to facilitate our interpretation in Table 3.3. We also introduce the notation h_not_ch(45) to denote h_choice(h_ch_range(0, 44), h_ch_range(46, 127), NULL) for ease of interpretation. h_not_ch(45) is not a valid Hammer function and is just used for notation. Often, character ranges just exclude one character, and this representation is more informative. To translate the parsers to English, we see that the first argument must be -Z followed by a nonempty argument and a third argument with cannot be -. The input -Z a b that crashed mkdir shown in Table 3.1 is included in this language.

In general, we play around with different sizes of symbolic arguments until we reach a minimum size and number of arguments that achieve high line coverage. This is indicated by the loop from the output of klee to symbolic arguments in Figure 3.1. With the GNU Coreutils library, we already have an idea of what inputs

arg00	h_sequence(h_repeat_n(h_ch('-'), 2), h_ch('c'), NULL)
arg01	h_ch_range(0, 127)
arg02	h_not_ch('-')

Table 3.4: Parser generated from running `mkdir` with a variable number of symbolic arguments, specifically letting there be 0 to 3 symbolic arguments of up to 3 characters. This differs from Table 3.3, which set each of its three symbolic arguments to fixed sizes of 2, 1 and 1.

trigger the vulnerability. In addition, the experiments in [9] found a framework of setting symbolic arguments to capture most behavior of the COREUTILS applications, specifically setting 3 symbolic arguments, one long and two short as well as a symbolic file, stdin and stdout. One of KLEE's limitations is that it seeks to find an input that triggers an error at the line, but not all such input. Different sizes and numbers of symbolic arguments can trigger the same error. Thus, even after catching an error and generating a set of constraints, as we did with `mkdir`, we can find other inputs that trigger the error with different sizes of symbolic arguments.

```
klee --optimize --libc=uclibc --posix-runtime ./mkdir.bc
--sym-args 0 3 3
```

The `--sym-args` option says to create from 0 to 3 symbolic arguments of up to 3 characters. The difference between this and our first call is before we had fixed 3 arguments of size up to 2, 1 and 1. Now, we have 0 to 3 arguments of size up to 3. Our tool outputs the parsers in Table 3.4, again with ASCII values substituted for readability.

The only difference in the constraints between this allocation of symbolic arguments is in the 0th argument. Allocating 3 instead of 1 character for the 1st and 2nd symbolic argument did not change its constraints, so we assume that the

current constraints hold for larger symbolic inputs. The 0th symbolic argument constraints changed from `-Z` to `--c`. Upon further inspection, the two different arguments denote the same command line option.

A symbolic argument of up to 3 characters should have also contained the 2 character input that triggered the error. However, once klee discovered a 3 character input (`--c`) that triggered the error, it did not describe different inputs that lead to the same line. klee's purpose is detecting the error and producing one input that triggers the error. However, several different types of inputs may trigger the same error. Testing different sizes of symbolic input to observe different constraints is still a manual process, but this can be automated. This multiplies the run time of the constraint generating process with multiple runs and taking the union of the constraints of errors. This is indicated by the arrow that goes from the program output back to adjusting symbolic arguments to see if different sizes of arguments trigger the same or different errors in Figure 3.1.

The constraints aren't perfect. Due to the nature of symbolic execution, the halting problem, an undecidable problem, is embedded in ensuring that every path is explored. We cannot prove or ensure every vulnerability is detected, so klee may encounter false negatives. We are unable to consistently discover all vulnerabilities in programs with klee, so we focus our discussion on defining parsers for detected vulnerabilities. Even then, klee constraints miss inputs that trigger the detected error the constraints are meant to capture. The symbolic execution engine does not generate complete constraints describing all input that trigger the error. For example in `mkdir`, the constraint of the 2nd argument not being '-' is still part of the language that crashes the program. This is not so much a false negative because it successfully found an error, but its constraints did not describe all inputs that led to it. The flip side of errors are false positives. We detect false positives by replaying

the inputs that triggered the generated error with `klee-replay` and verifying that the program crashed or otherwise failed.

The issue we address manually involves the true positives detected by klee. We want to capture all the input that trigger these errors. We begin the iterative process of running test cases by hand. Strings inside the grammar should crash the program or trigger the detectable vulnerability and strings outside the grammar should be correctly processed by the program. Informed by the constraints and knowledge of expected program behavior, we adjust the grammar and parser accordingly.

Ideally, we also would like to construct parsers for false negatives, as in errors undetected by klee, but klee does not give us any leads in that process. If we know of some vulnerability that is not being detected, we can add an assert statement and see if klee reports it as an assert error when running it on the updated program. We operate on the assert error constraints the same way as any other error.

We similarly repeat the experiment for the other programs with bugs in GNU Coreutils and report the outputted parsers in Table 3.5. Sometimes the order isn't as sensitive for different arguments and swapping them around will still cause the program to crash. For `mknod`, you can swap `arg00` and `arg01`. Because klee focuses on finding one input that triggers the bug, this swap is not reflected in the constraints and thus not in our automatically generated parser. The realization came from comparing the known input that triggered the error and the outputted grammar. This ability to swap two parameters will be represented by running `arg00` and `arg01` with both parser orders.

The parsers aren't complete in describing all input that trigger the errors because of the constraint generator of the tool klee. The intent of klee is not to generate constraints for every type of input that trigger the vulnerability, which would be ideal for our tool to then analyze. Because of this, the constraints are neither

mkdir, mkfifo	arg00	[h_sequence(h_repeat_n(h_ch('-'), 2), h_ch('c'), NULL); h_sequence(h_ch('-'), h_ch('Z'), NULL)]
	arg01	h_ch_range(0, 127)
	arg02	h_ch_range(0, 127)
mknod	arg00	[h_sequence(h_repeat_n(h_ch('-'), 2), h_ch('c'), NULL); h_sequence(h_ch('-'), h_ch('Z'), NULL)]
	arg01	h_ch_range(0, 127)
	arg02	h_ch_range(0, 127)
	arg03	h_ch('p')
md5sum	arg00	h_sequence(h_ch('-'), h_ch('c'), NULL)
	stdin	h_sequence(h_ch('M'), h_ch('D'), h_ch('5'), h_ch(' '), h_ch('('), NULL)
seq	arg00	[h_sequence(h_repeat_n(h_ch('-'), 2), h_ch('f), NULL), h_sequence(h_ch('-'), h_ch('f), NULL),]
	arg01	h_sequence(h_ch('%'), h_ch_range(1, 127))
	arg02	h_ch_range(1, 127)

Table 3.5: Parsers outputted by other runs on GNU Coreutils programs. If the parser is stored in a list, it signifies a union of these parsers.

comprehensive nor entirely accurate. We limit the false positives that affect the accuracy by replaying an input solved from the constraints, but that doesn't reflect all inputs in the constraints are not false positives. False positives may be due to bugs in klee or nondeterminism of the program. The lack of comprehensiveness is due to the fact that klee is more concerned that it detected an error than detecting all such formats that trigger the error. However, examples that trigger the same error usually follow a similar pattern. We can generalize the constraints for symbolic arguments restricted to 3 characters to more characters. Also, the constraints operate on a character by character basis, but this does not model many real world applications. We discuss limitations further in Chapter 4.

The outputted C file allows for easy customization of the outputted parsers to accommodate these errors. For example, with `seq`, we observe that the format option (`-f` or `--f`) throws the bug and the second parameter just needs to be a valid format string. If it isn't, the `seq` parser will throw a syntax error anyways. Then, we can rerun klee on `seq`, fixing the second symbolic input, with

```
klee --optimize --libc=uclibc --posix-runtime ./seq.bc
--sym-arg 3 %0 --sym-arg 2
```

We also run examples by hand. From the initial run of klee, we discover the error arose in the format option, so the first argument is fixed to `-f` or `--f`. The second parameter must consist of %, followed the a valid formatting sequence. Many improperly formatted arguments are detected by `seq`'s own parser like invalid floating point arguments, so to surpass their parser and crash the program, the input needs to be valid floating points. With these messages, we determine that the 1st argument should be a floating point format, and the 2nd argument should be a floating point. We construct the the parser by hand accordingly, as shown in Table 3.6. We use a few less intuitively named Hammer invocations, which are

described in Table 1.2.

arg00	[h_sequence(h_repeat_n(h_ch('-'), 2), h_ch('f), NULL), h_sequence(h_ch('-'), h_ch('f), NULL),]
arg01	h_sequence(h_ch('%'), h_optional(h_choice(h_ch('+'), h_ch('-'), NULL)), h_optional(h_sequence(h_many(h_ch_range('0', '9')), h_ch('.'), NULL)), h_many1(h_ch_range('0', '9')), NULL)
arg02	h_sequence(h_optional(h_choice(h_ch('+'), h_ch('-'), NULL)), h_optional(h_sequence(h_many(h_ch_range('0', '9')), h_ch('.'), NULL)), h_many1(h_ch_range('0', '9')), NULL)

Table 3.6: Modified parser for `seq`. `seq` prints a sequence of numbers from first to last, with first defaulting to 1. With knowledge of `seq`'s being triggered by the format option (`-f`), we implemented by hand `arg01` and `arg02` that would bypass `seq`'s parser for valid formats and crash the program.

Our tool's output from parsing klee's constraints gave us the outline of what is considered invalid input, but klee's constraints aren't powerful enough to describe all invalid input. Our tool successfully generates grammars and parsers from the constraints specified, but manual intervention is still required to achieve a more complete grammar. Due to limitations in computability, depending on the complexity of a language, it may be theoretically impossible to prove that the grammar we output is equal to the grammar of all inputs that do not trigger a vulnerability.

Section 3.2

BUSYBOX 1.10.2

We similarly apply our parser generating tool to errors detected in BUSYBOX 1.10.2 again on the same 64-bit Ubuntu Desktop. The BUSYBOX library provides a smaller version of common UNIX utilities, including programs in the COREUTILS library [1]. Cadar et. al also applied KLEE to find bugs in this library [9], and although not explicitly stated, we assume they ran the programs on a 32-bit machine like for COREUTILS. Again, we were only able to reproduce a subset of these errors and even fewer with running KLEE within a max time cutoff of 60 minutes. The programs of BUSYBOX come with a much less exhaustive set of tests, and that is indicated in the larger number of errors that KLEE finds per program. This introduced the need to run our tool on each of the constraints generated by the errors and take the union across all of them. We show parsers for 3 programs in BUSYBOX in Table 3.7.

We observe how several different inputs trigger errors on different lines in the program for `install` with a single symbolic argument. We take the union of all of the parsers generated by the various constraints in `arg00`. We also take the union between the two parsers listed in separate rows for `install`.

Upon expanding the number of symbolic arguments, we observe in `top` that the same characters, or some combination of them, trigger the error. In particular, a permutation of characters with ASCII values 0, 9, 10, 98, 100 and 110 for any number of symbolic arguments will trigger the error.

The constraints for `hexdump` combined to form any possible input. Running `hexdump` confirmed this behavior. However, `hexdump` behaves as expected if called on an existing file. This language cannot be expressed because KLEE does not have a concept of what files exist in the system.

install	arg00	[h_sequence(h_repeat_n(h_ch('-'), 2), h_ch('m'), NULL); h_sequence(h_repeat_n(h_ch('-'), 2), h_ch('g'), NULL); h_sequence(h_ch('-'), h_ch('g'), h_ch_range(0, 127), NULL)]
install	arg00	h_ch_range(0, 127)
	arg01	h_sequence(h_repeat_n(h_not_ch('-'), 2), h_not_ch('d'), NULL)
install	arg00	h_sequence(h_ch('-'), h_ch('d'), NULL)
	arg01	h_ch(0)
top	arg00, arg01, arg02, ...	h_many(h_choice(h_ch(0), h_ch(9), h_ch(32), h_ch('b'), h_ch('d'), h_ch('n'), NULL))
hexdump	arg00	h_ch_range(0, 127)

Table 3.7: Parsers outputted by runs on Busybox programs. If the parser is stored in a list, it signifies a union of these parsers.

We also looked at the most recent version of busybox, version 1.31.1 to see what errors still exist in these programs. klee only detected errors in `hexdump`, but upon replaying the input returned by the solver of these detected error-triggering constraints, we found them to be false positives. Although our tool was able to generate a parser for these constraints, we do not believe the constraints capture an error in the program. With more time, we would test our tool on other programs in the most recent busybox version.

Section 3.3

Generating Grammars from Parsers

We demonstrate another application for this tool with a simple example that serves as a proof of concept. Given a program that parses an input, we want to output the grammar for the input in the form of a parser combinator. The difference lies in the fact we do not want to protect against vulnerabilities, but instead capture the format of the input the program expects. There are currently limitations with using our tool to do this, especially regarding the complicated constraints generated by external function calls like `is_alpha`.

We wrote a function that parses a California license number to validate its format, specifically that it is 8 characters long, the first character is a letter, and the last 7 characters are digits between 0 and 9. The function consists of a series of nested *if*-statements that check that each character falls within the specified range, as shown in Figure 3.2. We insert an `assert(0)` statement in the inside of these *if*-statements to force KLEE to detect an error in this path. KLEE detects assert errors, among other errors, and outputs a set of constraints that lead to the error. Our tool parses these constraints and constructs a parser for these known invalid inputs. By taking a presumably correct parser, and making its success an error, our tool will generate a grammar for a successful input. Running KLEE generates the constraints that lead to the assert statement, which we then parse to construct the grammar.

If the program itself had a bug that KLEE detected, KLEE generates constraints for both the bug and the assert error. We can distinguish the two by checking which constraints had type assert error or what line number the error is thrown, both of which KLEE provides for every error.

The parser outputs a clean representation of the grammar:

Figure 3.2: Sample CA license plate parser to generate grammar

```
1    if 'a' <= num[0] && num[0] <= 'z') {
2      if ('0' <= num[1] && num[1] <= '9') {
3        if ('0' <= num[2] && num[1] <= '9') {
4            ...
5            if ('0' <= num[7] && num[1] <= '9') {
6                if (num[8] == '\0') {
7                    printf("Valid input!\n");
8                    assert(0);
9                }
10           }
11           ...
```

```
h_sequence(h_ch_range('a', 'z'), h_repeat_n(h_ch_range('0', '9'),
7), h_ch(0), NULL)
```

The technique of inserting an assert statement in a parser to extract the grammar can be applied to other applications. We can also fine tune our tool's constraint parser to be able to replicate more advanced grammars. As mentioned in the Related Work section, there exists prior work in synthesizing grammars from parser programs [16, 18].

Section 3.4

Contributions

We applied a symbolic execution tool, generally used for exhaustive testing, to generate grammars for LangSec. The goal of the grammar is to protect against detected vulnerabilities rather than capture the intent of the developers when writing the program. It operates on any program, not just parsers. Our tool begins constructing the grammar without any knowledge of concrete inputs, which none of the Related Work attempts.

We expanded the functionality of klee. While klee provides one example of an input that triggers the vulnerability, we construct a grammar for a set of inputs captured by the constraints and adjust the set with manual intervention. We output a parser implementation of the grammar, which can be prepended to the program to defend against the vulnerabilities detected by klee.

The parser file generated is easy to read and implemented in the Hammer parser combinator library. Because it is outputted in a compilable C file, it is also easy to adapt and customize with knowledge of the program or known counterexamples to the grammar. Recall that the outputted parser represents invalid input. When prepended to the program, it rejects input that it parses correctly, and allows input that it failed to parse through to the rest of the program.

We also showed that we can apply this to programs that validate input, with assert statements, to reconstruct the grammar. This is not to protect against vulnerabilities but to define the format of accepted input.

Chapter 4

Future Work

Section 4.1

Current Limitations

We first begin by considering the limitations of our tool in combination with KLEE and then discuss how they may be addressed.

Our tool currently operates on inputs as strings, treating each element in the symbolic array as a separate character. Our tool can be extended to integers with additional case analysis in our parsing of the constraint AST to invoke Hammer's `h_int` and `h_int_range`. We currently treat integers with each digit as a character.

Although the parsers we have generated cover the constraints that lead to the error, they are not comprehensive in parsing all strings that lead to the error. Generating more comprehensive parsers requires an iterative process of testing examples and counterexamples in the grammar and adjusting it accordingly. Based on results of the papers mentioned in the Related Work, this adaptive process of generating grammars can be automated for lower level languages [3,6]. For regular expressions and context free languages, a grammar can be formed and adjusted with a series of examples and counterexamples. We would alter our already formed

grammar with this process.

In addition, different sizes and numbers of symbolic inputs lead to different constraints, so a comprehensive grammar requires integrating the constraints of variable symbolic argument options. The user can input different symbolic arguments and take the union of all the parsers generated. This multiplies the run time by the number of different configurations of symbolic arguments.

Although symbolic execution intelligently traverses every path an input can take, any exhaustive search of all possible paths in a program will explode in memory and time. Optimizations can be applied, but this reduces the practicality for larger scale inputs and programs. However, the issue of path explosion is more a theoretical limit than an engineering problem.

We treat each symbolic input separately and assume that their constraints do not influence each other. For example, we assume there is no constraint saying that the first character in each symbolic argument is the same. Although this assumption is safe for the programs we tested our tool on, it may not be the case for an arbitrary program.

We discuss how some of these limitations can be addressed in future work.

Section 4.2

Modifying klee

The purpose of klee is to automatically perform tests that achieve high coverage on a diverse set of programs. While sufficient in detecting errors, klee only cares about discovering one input that triggers the error rather than the whole set of inputs. Thus, constraints are not comprehensive of possible input that may trigger the vulnerability. In order to capture a larger set of invalid input, a natural next step would be to modify the symbolic execution of klee.

Many tools are built on top of klee to achieve different purposes [10, 20, 24]. Ramos et. al configured klee to check the equivalence between two arbitrary klee functions [24]. Corin et. al extended the klee symbolic execution engine with a tainting mechanism to track information flows of data [10]. klee is a very powerful tool that we appended to by interpreting its constraints to generate a parser, but the method of generating these constraints should also be modified to best fit our needs. Future steps could modify klee to generate constraints which capture more inputs that lead to the given line rather than a subset of such inputs.

4.2.1. Directed Symbolic Execution

When we look at a program with a known vulnerability to establish a parser defending against this vulnerability, we usually know which exact line it occurs. Symbolic execution seeks to perform an exhaustive search of all possible paths, but in our use case, we only wish to explore the path that leads to the specified line. Ma et. al modified klee's search strategy to solve the problem of finding the shortest path to a certain line in the code [21]. This is relevant for programs with known buggy lines, so the symbolic execution can be directed to those specific lines. The vulnerable constraints can be more quickly and reasonably generated.

It will also be useful to tag areas of interest rather treat every line of code coverage equally. Future work in directed symbolic execution could weigh certain paths or lines above others in exploration priority. Bugs in lines of code that are not reached as often can be discovered with the ability to highlight these lines. In addition, it is infeasible to traverse every path of large programs, so directed symbolic execution narrows the focus.

4.2.2. Expanding System Environments

klee just models the file system, but other environments such as sockets or multi-threading are not supported. However, the inability to model the system environment is a known handicap for symbolic execution.

Section 4.3

Further Automation

After our tool generates a preliminary grammar, the grammar needs to be updated based on the knowledge of counterexamples. We can base future automation of synthesizing a grammar with examples and counterexamples on [3] and [6], which respectively tackle regular expressions and context free grammars. They rely on the ability to generate examples and counterexamples for a given grammar and check if these inputs are actually an example or counterexample of our conjectured grammar. The conjectured grammar is adapted if it doesn't contain an example, or if it fails to exclude a counterexample.

Section 4.4

Other Dynamic Analysis

The methods presented in [18] and [16] infer grammars from executing programs on a collection of given inputs. They work on programs that are intended to parse input to construct this grammar. Our task is distinct but related to theirs because we construct a grammar that protects against vulnerabilities. Perhaps a similar dynamic tainting process of the data can be applied to generate constraints. We can apply the tactics of memory tracing to a program whose intent is not to parse the input but still accesses and manipulates the input in some way.

These papers generate the grammar in Backus Normal Form. Another direction of automatically generating parsers is to take a grammar written in BNF and output a parser implementation of the grammar.

Section 4.5

Optimize Grammar

The current representation of the grammar in Hammer invocations is not the most readable or optimal form. For example, a sequence of characters can be converted to a token or a string. Also, after the union operation, the grammar can be further optimized and simply expressed.

Section 4.6

Testing Grammar

We currently test the grammar by checking whether known inputs that trigger the error and other inputs in similar formats are caught by the grammar, which is not a complete way of assessing the grammar. We can test the grammar with random fuzzing which is not a formal proof but has been a successful testing technique.

Chapter 5

Conclusion

The ultimate goal is to take a program as input and output a parser that protects against all vulnerabilities, not only known vulnerabilities. This will require the ability to detect the existence of any vulnerability, which given complex enough languages is theoretically impossible. We focus on generating a parser that protects against detectable vulnerabilities.

Our tool is able to automatically generate a parser combinator that rejects buggy inputs, as defined by constraints that lead to an error. Due to incompleteness in the constraints and limitations of symbolic execution in general, getting a satisfactory parser still requires manual intervention. For some programs, this intervention is nontrivial. With further efforts detailed in the Future Work chapter, the process can be further automated.

Using symbolic execution to generate grammars shows promise. Symbolic execution details what path is traversed to get to the vulnerable line, and the union of all such paths should inform what is considered valid or invalid input. The tool can also be used in conjunction with KLEE to get an idea of the types of inputs that trigger the error, which is helpful in the debugging process.

Bibliography

[1] Busybox. https://busybox.net/, May 2020.

[2] Coreutils. https://www.gnu.org/software/coreutils/, May 2020.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[4] Joao Antunes, Nuno Neves, and Paulo Verissimo. Reverse engineering of protocols from network traces. In *2011 18th Working Conference on Reverse Engineering*, pages 169–178. IEEE, 2011.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Notices*, volume 52, pages 95–110. ACM, 2017.

[7] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634, 2009.

[8] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329. ACM, 2007.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[10] Ricardo Corin and Felipe Andrés Manzano. Taint analysis of security code in the KLEE symbolic execution engine. In *International Conference on Information and Communications Security*, pages 264–275. Springer, 2012.

[11] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14, 2007.

[12] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402. ACM, 2008.

[13] Sheridan S Curley and Richard E Harang. Grammatical inference and machine learning approaches to post-hoc langsec. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 171–178. IEEE, 2016.

[14] Josh Fruhlinger. What is the heartbleed bug, how does it work and how was it fixed? https://www.csoonline.com/article/3223203/what-is-the-heartbleed-bug-how-does-it-work-and-how-was-it-fixed.html, 2017.

[15] E Mark Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[16] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Inferring input grammars from dynamic control flow. *arXiv preprint arXiv:1912.05937*, 2019.

[17] Dick Grune and Ceriel JH Jacobs. Parsing techniques. *Monographs in Computer Science. Springer,*, page 13, 2007.

[18] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725. IEEE, 2016.

[19] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[20] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. *ACM SigPlan Notices*, 48(10):19–32, 2013.

[21] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011.

[22] Michaël Mera. Mining constraints for grammar fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–418, 2019.

[23] Falcon Momot, Sergey Bratus, Sven M Hallberg, and Meredith L Patterson. The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 45–52. IEEE, 2016.

[24] David A Ramos and Dawson R Engler. Practical, low-effort equivalence verification of real code. In *International Conference on Computer Aided Verification*, pages 669–685. Springer, 2011.

[25] Len Sassaman, Meredith L Patterson, Sergey Bratus, and Michael E Locasto. Security applications of formal language theory. *IEEE Systems Journal*, 7(3):489–500, 2013.

[26] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.